Implementation and Complexity of the Lowest Static Reduction

Ludovic Henrio, Bernard Paul Serpette, Szabolcs Szentes

▶ **To cite this version:**

Ludovic Henrio, Bernard Paul Serpette, Szabolcs Szentes. Implementation and Complexity of the Lowest Static Reduction. RR-5034, INRIA. 2003. inria-00071550

HAL Id: inria-00071550
https://inria.hal.science/inria-00071550

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implementation and Complexity of the Lowest Static Reduction

Ludovic Henrio — Bernard Paul Serpette — Szabolcs Szentes

N° 5034

5 décembre 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--5034--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Implementation and Complexity of the Lowest Static Reduction

Ludovic Henrio , Bernard Paul Serpette , Szabolcs Szentes

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oasis

Rapport de recherche n° 5034 — 5 décembre 2003 — 17 pages

Abstract: The lowest static reduction (*LSR*) is the 0^{th}-order Control-Flow Analysis without continuation passing style (*CPS*) consideration. This article presents some algorithms resolving LSR and their related complexities. Even if this analysis is generally declared to be computable in cubic time, a stand alone algorithm reaching this complexity is not so straightforward. With the λ-calculus as input language, we will start with a *blind* algorithm and step by step we will finally exhibit the algorithm which run, for worst cases, in cubic time related to the size of the program.

Key-words: Static analysis, static reduction, 0CFA, complexity

This report corresponds to a work launched by Szabolcs Szentes in 2002 and extended by Ludovic Henrio in 2003. This article was submitted and accepted at the JFLA'04 (*Journées Francophones des Langages Applicatifs*), and thus has been translated in french. Here is the original version written in (surely bad) english.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Algorithmes et complexités de la réduction statique minimale

Résumé : La réduction statique minimale (*RSM*) est l'analyse de flot de contrôle d'ordre 0 (*0CFA*). Cet article présente une série d'algorithmes résolvant cette analyse ainsi que leurs complexités respectives. Même si cette analyse est souvent annoncée comme étant calculable en un temps cubique par rapport à la taille du programe étudié, un algorithme complet atteignant cette complexité n'est pas si évident. Avec le λ-calcul comme langage de description des programmes, nous commençons par un algorithme naïf et, pas à pas, nous aboutissons à celui qui s'exécute, dans le pire des cas, en un temps cubique. Nous donnerons les exemples qui prouvent que ces complexités maximales peuvent être atteintes.

Mots-clés : Analyse statique, réduction statique, 0CFA, complexité

1 Introduction

The analysis described in this article calculates, for each Abstract Syntax Tree (AST) node of a program, a set of values that this node may compute at runtime. This analysis was first studied in the abstract interpretation domain: [18, 19, 20, 16, 2, 17, 11, 4, 1]. Then, Nevin Heintze initiated a work based on constraint resolution: [5, 14, 9, 8, 10, 6, 13]. More recently, this analysis was studied with the type theory point of view: [15, 7, 12, 21].
In this paper, in order to analyze the complexity, we have tried to explicit as far as possible the resolution algorithm. In other words, we have pulled up in the algorithm what an abstract interpreter, a constraints solver or a type checker is intended to do.
Besides the technology used for this analysis, there is a wide variety of strategies concerning the tradeoff between accuracy and complexity. For example, the Static Reduction Analysis (SRA [16]) ranges from an exact analysis which is a collecting evaluator, to a simple analysis, which is related to the 0^{th}-order Control-Flow Analysis (OCFA [19]) or can also be viewed as the Set Based Analysis (SBA [5]).
Since we aim at comparing different algorithms, it is necessary to have a clean, simple, and well-defined specification of the analysis: all algorithms must compute the same result. This is why we have chosen the Lowest Static Reduction (LSR) for our study: for the λ-calculus, LSR is defined as the smallest function satisfying only three simple rules.
The main contribution of this paper is to propose algorithms *ready to implement* for an *open language* (no Continuation Passing Style (CPS) restriction). The only operators left undefined are $\forall x \in S : P$ (a.k.a. **map**) and set functions $(\in, \subset, \cup)$. With bit vectors, $\in$ can be done in constant time while $\subset$ and $\cup$ may have a linear complexity.
The article is organized as follows. Section 2 presents the context and the LSR analysis. Section 3 presents four different algorithms calculating the LSR and their complexity. Section 4 presents some killer series that illustrate the worst case of complexity of preceding algorithms. Section 5 discusses about the 0CFA specification and some future work.

2 Definitions

This section defines what the algorithms described in this paper are intended to resolve. First, we give the language analyzed by the algorithm. For simplicity we restrict this language to be the λ-calculus; in [3], the analysis is extended to an imperative object-oriented language. Then, we briefly give the general definition of the Static Reduction Analysis (SRA). And the lowest analysis (LSR) is defined as an instantiation of SRA. Finally, we will give general equations concerning the size of a program.

2.1 Input language

The input language, namely the λ-calculus, is described by the following abstract syntax:

$$\begin{array}{lcl} Expr & = & Lambda \cup Var \cup App \\ Lambda & = & \{\lambda\} \times Var \times Expr \\ App & = & Expr \times Expr \end{array}$$

The type $Expr$ is parameterized over a set of variables named Var. The singleton $\{\lambda\}$ is introduced in the type $Lambda$ in order to avoid $Lambda \subseteq App$. The elements of $Lambda$ will be noted $\lambda x.B$ and the elements of App will be noted $(F\ A)$. For this simple language, values are functions: $Value = Lambda$.

2.2 Generic SRA

The generic SRA is parameterized by an **approximation function** $\mathcal{F}$ whose signature is $App, Lambda, Value \rightarrow Lambda$. This approximation function must return an abstraction which is α-equivalent (equality modulo bound variables renaming) to its second argument: $\mathcal{F}(e, l, a) \equiv_\alpha l$.
Then, the static reduction is defined as the **smallest** relation from expressions to values $(Expr \rightarrow Value)$ satisfying the following rules:

$$\frac{l \in Lambda}{l \rightarrow l}$$

$$\frac{F \rightarrow f \ \wedge \ A \rightarrow a \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a)}{x \rightarrow a}$$

$$\frac{F \rightarrow f \ \wedge \ A \rightarrow a \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a) \ \wedge \ B \rightarrow v}{(F\ A) \rightarrow v}$$

The static reduction can also be viewed as a function $\mathcal{S}$ from expressions to sets of values $(Expr \rightarrow \mathcal{P}(Value))$ where, by definition: $v \in \mathcal{S}(e) \Leftrightarrow (e \rightarrow v)$. $\mathcal{S}$ denotes the successors in the graph induced by the relation $\rightarrow$. Thus, $\mathcal{S}$ is the smallest function satisfying the rules:

$$\frac{l \in Lambda}{l \in \mathcal{S}(l)}$$

$$\frac{f \in \mathcal{S}(F) \ \wedge \ a \in \mathcal{S}(A) \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a)}{a \in \mathcal{S}(x)}$$

$$\frac{f \in \mathcal{S}(F) \ \wedge \ a \in \mathcal{S}(A) \ \wedge \ \lambda x.B = \mathcal{F}((F\ A), f, a) \ \wedge \ v \in \mathcal{S}(B)}{v \in \mathcal{S}((F\ A))}$$

The approximation functions are in a lattice whose lowest element is $\mathcal{F}_\perp(e, l, a) = l$. This function defines the lowest static reduction (LSR) which will be analyzed throughout this paper.

2.3 Lowest Static Reduction

With the approximation function $\mathcal{F}_\perp(e,l,a) = l$, one can see that the overall analysis only manipulates sub-expressions of the main program. This set, denoted by $\mathcal{E}$, is bounded and known as long as the main expression is known. Following the projection on Var, *Lambda* and *App*, we can make a partition of $\mathcal{E}$: $\mathcal{E} = \mathcal{V} \oplus \mathcal{L} \oplus \mathcal{A}$. Therefore, the rules of the generic SRA can be simplified and become:

$$\frac{l \in \mathcal{L}}{l \in \mathcal{S}(l)} \tag{1}$$

$$\frac{(F\ A) \in \mathcal{A} \ \wedge\ \lambda x.B \in \mathcal{S}(F)}{\mathcal{S}(A) \subseteq \mathcal{S}(x)} \tag{2}$$

$$\frac{(F\ A) \in \mathcal{A} \ \wedge\ \lambda x.B \in \mathcal{S}(F)}{\mathcal{S}(B) \subseteq \mathcal{S}((F\ A))} \tag{3}$$

2.4 Size and position

All the complexities given in this article are related to the *size* N of the main expression. We will take this size as the cardinal of the set $\mathcal{E}$: $N = \|\mathcal{E}\|$. This is not the standard way of counting sub-expressions of an expression: generally each use of a variable count for one. For us the size of the expression $\lambda x.(x\ x)$ is 3: One abstraction, one application and one variable. Therefore the AST is not really a tree since all occurences of the same variables are merged (a formal parameter and all its uses).
The notion of *position* is the inverse of the AST. For example, if $E = (F\ A) \in \mathcal{A}$, $E \to F$ is an edge in the AST and we state that E is a position of F. Since the AST is no more a tree, variables may have more than one position. Hence, the Pos function, denoting the position of a node, has the signature $\mathcal{E} \to \mathcal{P}(\mathcal{E})$. However, abstractions and applications cannot have more than one position: $\forall e \in (\mathcal{E} - \mathcal{V}) : \|Pos(e)\| \leq 1$. Only the main program and unused variables may have no positions.
In the λ-calculus, a node X can be in the function position of an application node (i.e. $(X\ ?)$), or in the argument position of an application node (i.e. $(?\ X)$), or in the body position of an abstraction (i.e. $(\lambda ?.X)$). The abstraction $\lambda x.B$, is not a position of the variable x (unless B is x).
Following the projection of expressions on variables, abstractions and applications we have: $\|\mathcal{E}\| = N = \|\mathcal{V} \oplus \mathcal{L} \oplus \mathcal{A}\| = \|\mathcal{V}\| + \|\mathcal{L}\| + \|\mathcal{A}\| = N_\mathcal{V} + N_\mathcal{L} + N_\mathcal{A}$. Without free variables we have $N_\mathcal{L} = N_\mathcal{V}$.
With these sizes we can list some useful inequations. Since values are abstractions: $\forall e \in \mathcal{E} : \|\mathcal{S}(e)\| \leq N_\mathcal{L}$. Then, knowing that a graph and its inverse have the same number of edges, the sum of positions length is the number of edges of the AST: $\sum_{e\in\mathcal{E}} \|Pos(e)\| = \sum_{e\in\mathcal{E}} \|Pos^{-1}(e)\| = N_\mathcal{L} + 2N_\mathcal{A}$.

We can conclude with an inequation which will be used in complexities calculus [1]:

$$\sum_{e \in \mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\| \leq N_{\mathcal{L}}(N_{\mathcal{L}} + 2N_{\mathcal{A}}) \leq 2N^2$$

3 Algorithms

This section describes four algorithms calculating $\mathcal{S}$. The *result graph* obtained by those analyses has $\mathcal{O}(N)$ *nodes* which are the nodes of the expression and potentially $\mathcal{O}(N^2)$ *edges* linking nodes to *Values* (i.e. Lambdas).
The first algorithm is a strict implementation of the LSR rules. We will show that this algorithm has a potential $\mathcal{O}(N^5)$ complexity, the section 4 will give expressions for which the analysis gets at this complexity. The second algorithm, also described in [17], is a specialization of the first one where application nodes are taken in a specific order. This evaluation-driven algorithm has also an $\mathcal{O}(N^5)$ complexity. The third algorithm is an optimization where a virtual time is added to each node's value. This time allows some shortcuts between fix-point iterations. This time-sorted values algorithm has a $\mathcal{O}(N^4)$ worst case complexity. The last algorithm, when adding a new value to a node, makes specific cases with the position of this node, this algorithm is in its spirit the one which is described in [2]. At last, this position-driven algorithm reach the $\mathcal{O}(N^3)$ worst case complexity.

3.1 Blind algorithm

```
0:0  ∀l ∈ L : S(l) ← {l}                                  Rule 1
1:0  more ← true                                          Fix-point initialization
2:2  while(more)                                          Fix-point iteration
3:2      more ← false                                     Fix-point reset
4:2      ∀(F A) ∈ A :                                     Take all application nodes
5:3          ∀λx.B ∈ S(F) :                               Take all abstractions of the function part
6:4              Update(x, A)                             Check rule 2
7:4              Update((F A), B)                         Check rule 3

8:4  Update(dst, src)
9:4      if (S(src)⊆S(dst)) return                        The rule is already satisfied
10:2      S(dst) = S(dst) ∪ S(src)                        Update S to satisfy the rule
11:2      more ← true                                     At least one edge is added
```

Figure 1: Blind algorithm

[1] With more accurate analysis we can fall down this upper bound to $2N_{\mathcal{L}}(N_{\mathcal{A}} + 1)$

The first and simplest algorithm is a direct implementation of LSR rules. First we apply rule 1 on all abstractions. Then for each application and abstraction satisfying premises of rules 2 and 3, we check corresponding conclusions adding edges if necessary. A fix-point is reached when no more edge can be added. This algorithm is described in the Figure 1. Each line starts with n:**k** where n is a line number and **k** is the degree of execution of the line in the worst case complexity (i.e. n:**k** means, in the worst case, the line n is executed $\mathcal{O}(N^k)$ times).
In the worst case, the result graph must be *dense*: the graph has $\mathcal{O}(N^2)$ edges, i.e. $\sum_{e\in\mathcal{E}} \|\mathcal{S}(e)\| = \mathcal{O}(N^2)$. A nasty program leads to an analysis where a bounded number of edges is added at each iteration and the final result graph is dense. Such a program imposes that the number of needed fix-point iterations is proportional to N^2 . Hence, lines *2*, *3*, *4*, *10* and *11* may be executed $\mathcal{O}(N^2)$ times. If application nodes are on the same order as N ($N_{\mathcal{A}} = \mathcal{O}(N)$), then the line *5* is executed $\mathcal{O}(N^3)$ times. If the subgraph induced by $\mathcal{A}$ is also dense ($\sum_{(F\ A)\in\mathcal{A}} \|\mathcal{S}(F)\| = \mathcal{O}(N^2)$)[2] the line *6* and *7* are executed $\mathcal{O}(N^4)$ times. Thus, the *Update* function may be called $\mathcal{O}(N^4)$ times. Since the complexity of the set inclusion predicate is at least linear, the overall complexity[3] is $\mathcal{O}(N^5)$. The killer series given in section 4 raise this complexity.
The main drawbacks of this blind algorithm are:

1. Application nodes of $\mathcal{A}$ are taken in a predefined order. It will be beneficial to visit callee nodes before caller's one. This will be the improvement of the *evaluation-driven* algorithm.

2. Between iterations it doesn't memorize on which nodes the *Update* function was called. It is easy to see that if $Update(dst, src)$ is called twice on the same nodes, and if no value was added to $\mathcal{S}(src)$ between the two calls, then the second call to *Update* is not necessary since we can prove that $\mathcal{S}(src)\subseteq\mathcal{S}(dst)$ is true. This improvement will be provided by the *time-sorted* algorithm.

3. When an edge is added, it does not try to compute the set of application nodes which may be influenced by this new edge. This improvement will be proposed in the *position-driven* algorithm.

3.2 Evaluation-driven algorithm

We will see in section 4 that, in the blind algorithm, the number of iterations is influenced by the order in which application nodes are taken: we can expect better results when these application nodes are taken in a dedicated order. Intuitively, like for an evaluator, it seems better to follow the body of a function before going up at the application node where this function is applied. For ensuring termination of the algorithm in case of recursion, we still need a fix-point resolution and analyze each function body only once for each iteration.

[2] That is obtained when for $\mathcal{O}(N)$ application nodes $(F\ A)$, $\|\mathcal{S}(F)\| = \mathcal{O}(N)$
[3] The real complexity is $\mathcal{O}((N_{\mathcal{V}} + N_{\mathcal{A}})N_{\mathcal{L}}N_{\mathcal{A}}N_{\mathcal{L}}N_{\mathcal{L}})$

```
0:0  ∀l ∈ L : S(l) ← {l}                          Rule 1
1:0  time ← 0                                     Time initialization
2:0  ∀l ∈ L : Time(l) ← −1                        Map Time initialization
3:0  more ← true                                  Fix-point initialization
4:2  while(more)                                  Fix-point iteration
5:2      more ← false                             Fix-point reset
6:2      Eval(Main)                               Abstract evaluation of main expression
7:2      time ← time + 1                          Time increment

8:3  Eval(e)                                      Abstract evaluation of e
9:3      if (e = (F A) ∈ A)                       Only application nodes have to be updated
10:3         Eval(F)                              Evaluate the function
11:3         Eval(A)                              Evaluate the argument
12:3         ∀f = λx.B ∈ S(F) :                   For each function f
13:4             Update(x, A)                     Check rule 2
14:4             if Time(f) ≠ time                Function body analyzed once by iteration
15:3                 Time(f) ← time               Put the time on this function
16:3                 Eval(B)                      Evaluate the body
17:4             Update(e, B)                     Check rule 3
```

Figure 2: Evaluation-driven algorithm

```
0:0  ∀l ∈ L : S(l) ← (< l, 0 >)                                  Rule 1
1:0  time ← 0                                                    Time initialization
2:0  more ← true                                                 Fix-point initialization
3:2  while(more)                                                 Fix-point iteration
4:2      more ← false                                            Fix-point reset
5:2      ∀(F A) ∈ A :                                            Take all application nodes
6:3          ∀ < λx.B, t >∈ S(F) :                               Take all abstractions of the function part
7:4              Update(x, A, t ≥ time)                          Check rule 2
8:4              Update((F A), B, t ≥ time)                      Check rule 3
9:2      time ← time + 1                                         Time increment

10:4 Update(dst, src, new?)
11:4     if new?                                                 For new functions
12:2         ∀ < v, ⋆ >∈ S(src) :                                Check all values of src
13:3             UpdateValue(dst, v)                             Update one value
14:4     else                                                    For old functions
15:4         ∀ < v, t' >∈ S(src) with t' ≥ time :                Check new values
16:3             UpdateValue(dst, v)                             Update one value

17:3 UpdateValue(dst, v)
18:3     if (< v, ⋆ >∈ S(dst)) return                            The value is already added
19:2     S(dst) = {< v, time + 1 >} ∪ S(dst)                     Update S
20:2     more ← true                                             One edge is added!
```

Figure 3: Time-sorted algorithm

This evaluation-driven algorithm is given in Figure 2. At each iteration a variable *time* is incremented. Each abstraction has its own time annotation (updatable map $Time$ from $\mathcal{L}$ to integer); $Time(f) = n$ means that the body of the abstraction f was analyzed in the n^{th} iteration. The function *Update* is the one defined for the blind algorithm (Figure 1).
One can say that this algorithm was made popular by Shiver's thesis. We will discuss variations of this algorithm in section 5. One may notice that this algorithm *fully* computes the LSR only if the input program doesn't contains dead code.
The same complexity discussion as the one done for the blind algorithm can be applied to this evaluation-driven algorithm. If $\mathcal{O}(N^2)$ fix-point iterations are performed (line *4* to *7*), and without dead code, the function *Eval* is called $\mathcal{O}(N^3)$ times (in each iteration, one time for $\mathcal{O}(N)$ node). With $\sum_{(F\ A)\in\mathcal{A}} \|\mathcal{S}(F)\| = \mathcal{O}(N^2)$) then the *Update* function is called $\mathcal{O}(N^4)$ times and thus the eval-driven algorithm has also a $\mathcal{O}(N^5)$ worst case complexity.

3.3 Time-sorted values algorithm

In order to optimize the number of calls to the function *Update* in the previous algorithms, we annotate edges by the iteration number, viewed as a virtual time, in which the edge is added. At a time T, we will associate the time $T+1$ for each added edge. Thus, at a time T, a value of a node is called *new* if its time is greater or equal than T, and called *old* otherwise. We can check that the analysis of an application node is influenced by:

1. the new argument values. These values must be propagated to parameters of **all** function values: old function values must be warned of these new arguments.
2. the new function values. **All** argument values must be propagated to the new parameters and **all** body values of these new functions must be propagated to the application node.
3. all function bodies values. Unfortunately, **all** function values must be fetched in order to retrieve new body values that must be propagated to the application node.

What we have gained is that old argument values are not checked against old function values. This time-sorted algorithm is described in Figure 3. The new *Update* function receives a flag indicating if the rule must be checked for a *new* function. If so, as previous algorithms, all source value are visited. For old function values, only new source values are checked.
To calculate the complexity, one can observe that, when adding a value v to the node X at time t, each position P of X may trigger at most $2 * N$ calls to $UpdateValue$ at time $t+1$. Indeed, following the description of the algorithm, let us perform a case analysis on the position P.

1. $P = (F\ X)$. The new value v causes at most N calls to $UpdateValue$ at time $t+1$: one call for each (at most N) function of F.
2. $P = (X\ A)$. The new function v triggers less than $2N$ calls at time $t+1$ due to the two calls to *Update*.
3. $P = \lambda y.X$. The new value v causes at most N calls to $UpdateValue$ at time $t+1$. One call for each case (at most N) where $\lambda y.X$ is a functional value of an application.

If we note n_t^e the number of values added to e at time t, u_t the number of $UpdateValue$ calls at time t, then we just proved that: $u_{t+1} \leq 2N \sum_{e \in \mathcal{E}} n_t^e \times \|Pos(e)\|$. If we note T the number of fix-point iterations (i.e. the last value of $time$), then:

$$\begin{aligned}\sum_{t \leq T} u_{t+1} &\leq \sum_{t \leq T} 2N \sum_{e \in \mathcal{E}} n_t^e \times \|Pos(e)\| \\ &= 2N \sum_{e \in \mathcal{E}} (\sum_{t \leq T} n_t^e) \times \|Pos(e)\| \\ &= 2N \sum_{e \in \mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\| \\ &\leq 4N^3 \quad \text{(see section 2.4)}\end{aligned}$$

The term $\sum_{t \leq T} u_{t+1}$ represents the number of time the algorithm reach the line *17*, which have the same order than lines *13*, *16* and *18*. For this last line, we suppose that there is a

```
00:0  ∀l ∈ L : UpdateValue(l, l)                          Start the recursion by Rule 1

01:3  UpdateValue(dst, v)                                 Add value v in node dst
02:3      if (v ∈ S(dst)) return                          Check if already added
03:2      S(dst) ← S(dst) ∪ {v}                           Add the value
04:2      ∀P ∈ Pos(dst) : UpdatePos(P, dst, v)            Update each position

05:2  UpdatePos(P, dst, v = λx.B)                         Update by position
06:2      if (P = (dst A) ∈ A)                            Case 1
07:2          Redexes(v) ← Redexes(v) ∪ {P}                Update redexes
08:2          ∀(a ∈ S(A)) : UpdateValue(x, a)
09:2          ∀(r ∈ S(B)) : UpdateValue(P, r)
10:2      if (P = (F dst) ∈ A)                            Case 2
11:2          ∀(λy.C ∈ S(F)) : UpdateValue(y, v)
12:2      if (P = λz.dst ∈ L)                             Case 3
13:2          ∀(E ∈ Redexes(P)) : UpdateValue(E, v)
```

Figure 4: Position-driven algorithm

constant time algorithm for testing that a value belongs to a set, this can be achieved with variations on bit-vectors.
Due to bounded edges of the graph, lines *19* and *20* cannot be more than $\mathcal{O}(N^2)$. Complexity analysis of the blind algorithm said that all other lines cannot be more than $\mathcal{O}(N^4)$, thus we can conclude that the worst case complexity of this time-sorted algorithm is $\mathcal{O}(N^4)$.
For the lines *12* and *15* we give the result of the killer series of section 4. We can note that line *12* can raise to 4 (but then most $\mathcal{S}(src)$ must be empty) or 3 ($\mathcal{S}(src)$ contains a constant number of values). This consideration is also valid for line *15* where the killer series reach the $\mathcal{O}(N^4)$, but with a bit-vector implementation this line may hide a $\mathcal{O}(N^5)$ complexity (the whole vector must be scanned to know that it doesn't contains a new value), one can solve this problem with both a bit-vector and a time-sorted list for node values (i.e. line *19* put a bit in the vector and add the new value in front of the list).
Considering the blind algorithm, we have gained one order of magnitude by canceling some calls to *Update* that have already been performed . This algorithm is $\mathcal{O}(N^4)$ only because $\mathcal{O}(N^2)$ iterations may be performed. For $\mathcal{O}(N)$ iterations, the complexity falls down to $\mathcal{O}(N^3)$.

3.4 Position-driven algorithm

The complexity analysis of the time-sorted algorithm suggests that, when adding a new value to a node,the position of this node is crucial. This induces the position-driven algorithm

given in Figure 4. Let's analyze the consequences of adding a new value $v = \lambda y.B$ to a node X at a position P.

1. When $P = (X\ A)$, the formal parameter y may receive some new values via A, and P may receive new values via the body B.
2. When $P = (F\ X)$, formal parameters of all values of F may receive this new value v.
3. When $P = \lambda z.X$, every application node where this abstraction is applied may receive this new value v. This set of application nodes is incrementally constructed by the function *Redexes* ($\mathcal{L} \rightarrow \mathcal{P}(\mathcal{A})$). This function is updated each time a new value is added to the function part of an application (namely in the case 1).

This algorithm is in the spirit of the one described in Ayers's thesis [2]. It highlights what a constraints solver is intended to do: resolving equations by following a dependence graph. Here this dependence graph is directly coupled with the *position* notion. We can note that Ayers's algorithm is applied on programs in CPS form. In this context, values of intermediate application nodes in the source program are transferred to a variable in a generated continuation. Hence, for CPS programs, the analysis doesn't have to compute values of application nodes, therefore the third case is not necessary and Ayers's algorithm don't have to construct the *Redexes* function.
For complexity we start with the line *3* and *4* which are executed $\sum_{e\in\mathcal{E}} \|\mathcal{S}(e)\|$ times. Then the complexity of the line *5* is the number of calls performed line *4* which is exactly $\sum_{e\in\mathcal{E}} \|\mathcal{S}(e)\| \times \|Pos(e)\|$. Knowing that term is $\mathcal{O}(N^2)$ (see section 2.4), then all lines of the *UpdatePos* function have at most this complexity. Since each one of these line is performed in linear time, then the number of time that the function *UpdateValue* is called is $\mathcal{O}(N^3)$. Thus the overall complexity is $\mathcal{O}(N^3)$.

4 The Killer Series

All the series developed in this section, and the degree of complexity related to each algorithm are summarized in Figure 5. For the blind algorithm, we have tested three different ways of collecting application nodes: *fac* means that the application node (c) appears **after** its children in the AST (the function f and the argument a), *cfa* is not really intuitive since application nodes are visited before their childrens, and finally *cfac* is the combination of both where application nodes are visited twice in an iteration. We will see that the latter does not always have the minimum complexity of the first two.
In order to simplify the description of the series, here are common combinators:

$I \equiv \lambda x.x$
$True \equiv \lambda x.\lambda y.x$
$False \equiv \lambda x.\lambda y.y$
$\Delta \equiv \lambda x.(x\ x)$
$Y \equiv \lambda m.(\Delta\ \lambda f.(m\ \lambda x.(f\ f\ x)))$

function	**blind**			**eval**	**time**	**position**
	fac	cfa	cfac			
Base	3	3	3	2	2	1
BaseRec	3	3	3	3	2	2
I	4	4	4	3	3	**3**
I-chain	4	**5**	4	4	3	**3**
Deep-let	**5**	**5**	4	4	**4**	**3**
Deep-let2	**5**	**5**	**5**	**5**	**4**	**3**
Roll-left	**5**	4	4	**5**	**4**	**3**
Roll-right	4	**5**	4	4	3	**3**

Figure 5: Order of complexity for the killer series

Y is the call-by-value fix-point combinator used by **letrec**. Here are some common syntactic sugars:

$\lambda x_1 x_2 \ldots x_n.B \equiv \lambda x_1.\lambda x_2.\ldots.\lambda x_n.B$
$(F A_1\ A_2 \ldots A_n) \equiv (\ldots((F\ A_1)\ A_2)\ldots A_n)$
$A;B \equiv (False\ A\ B)$
$(\textbf{if}\ P\ A\ B) \equiv (P\ A\ B)$
$(\textbf{let}\ x = V\ \textbf{in}\ B) \equiv (\lambda x.B\ V)$
$(\textbf{letrec}\ f = V\ \textbf{in}\ B) \equiv (\textbf{let}\ f = (Y\ (\lambda f.V))\ \textbf{in}\ B)$

First we need N values, which means N functions. The simplest way to do that is to chain these functions: $\lambda x_N.\ldots.\lambda x_1.\ <Body>$. For simple, one can take the identity function for the body we have: $V_N = \lambda x_N \ldots x_1.I$. We can say that each function *hides* its predecessor, in the same way as a pointer hides its value. The only way to reveal this hidden value is to apply the function, which can be done, in the shortest way, by using Δ. Thus to retrieve the latest body (i.e. I) we have to generate:

$$\text{Base}_N \equiv (\Delta_N \ldots (\Delta_1\ V_N)\ldots)$$

This simple expression reaches two orders below the worst case complexity for every algorithm except the eval-driven one for which we have only $\mathcal{O}(N^2)$. This expression doesn't create a dense graph. Such a graph can be obtained by merging all Δ_i in a loop:

$$\text{BaseRec}_N \equiv (\textbf{letrec}\ loop = \lambda v.(\textbf{if}\ ?\ v\ (loop\ (\Delta\ v)))\ \textbf{in}\ (loop\ V_N))$$

Where ? is any expression which produce the true and false booleans, for example ? $\equiv$ $(\textbf{let}\ f = I\ \textbf{in}\ (f\ True);(f\ False))$. The BaseRec series have a constant number of application nodes and thus do not modify the complexities showed by $Base_N$ except for the position-driven and the eval-driven algorithms.

The last stage for producing killer series is to insert a call to a common function F_N between each Δ extraction of Base_N:

$$\text{Pattern}_N \equiv (\textbf{let}\ f = F_N\ \textbf{in}\ (\Delta_N\ (f\ \ldots(\Delta_1\ (f\ V_N))\ldots)))$$

The F_N function is intended to use its parameter value and to send it back to the next Δ application. Since F_N appears only once in Pattern$_N$, its size may be in $\mathcal{O}(N)$. But we can begin with a constant size function ($F_N = I$) which strangely reveals the worst case complexity of the position-driven algorithm. Next, a simple way to have a N sized identity function is to call the identity N times:

$$F_N\text{=I-chain}_N \equiv \lambda x.(I_N \ \ldots \ (I_1 \ x)\ldots)$$

This series reveals the worst case for the blind algorithm when application nodes are taken in a reverse order. It shows that this order may produce different order of complexities. In this series, application nodes are in argument position. To switch application nodes in function position it suffices to generate a chain of redexes, or equivalently to produce a cascade of **let**:

$$\begin{aligned} F_N\text{=Deep-let}_N \quad &\equiv \lambda x.(\lambda x_N.(\lambda x_{N-1}.(\ldots((\lambda x_1.x_1)\ x_2)\ldots)\ x_N)\ x) \\ &\equiv \lambda x.(\textbf{let}\ x_N = x\ \textbf{in}\ (\textbf{let}\ x_{N-1} = x_N\ \textbf{in}\ \ldots(\textbf{let}\ x_1 = x_2\ \textbf{in}\ x_1)\ldots)) \end{aligned}$$

This way, the deepest redex (**let** $x_1 = x_2$ **in** x_1) operates on a not yet analyzed argument. This series can also be given by induction:

$$\begin{array}{lll} U_1 & \equiv \lambda x1.x1 & \\ U_i & \equiv \lambda x_i.(U_{i-1}\ x_i) & (i > 1) \\ \text{Deep-let}_N & \equiv \lambda x.(U_N\ x) & \end{array}$$

This series reveals the worst case complexities for all algorithms but the eval-driven and blind when application nodes are visited twice (once before, once after its children). Remind that the evaluation-driven algorithm is blocked in its walk when the same function is called twice. To do so, each function U_i has to be called twice, first with a dummy argument (say I), then with the correct one. One can define the function Deep-let2$_N$ with this definition of U_i:

$$U_i \equiv \lambda x_i.(\textbf{let}\ f = U_{i-1}\ \textbf{in}\ (f\ I);(f\ x_i)) \qquad (i > 1)$$

With this series, all algorithms reach their worst case complexity. A less tortuous series which kills the eval-driven algorithm can be found in Ayers's thesis [2], instead of calling twice each functions, one can use recursion and roll the values:

$$F_N\text{=roll-left}_N \equiv \lambda x.(\textbf{letrec}\ R = \lambda x_1 x_2 \ldots x_N.(\textbf{if}\ ?\ x_1\ (R\ x_2 \ldots x_N\ x_1))\ \textbf{in}\ (R\ I \ldots I\ x))$$

This series has the same effect that Deep-let2$_N$ except for the blind algorithm when application nodes are taken after their children. However, the direction of rolling arguments is important, one can define our last series with:

$$\text{roll-right}_N \equiv \lambda x.(\textbf{letrec}\ R = \lambda x_1 \ldots x_{N-1} x_N.(\textbf{if}\ ?\ x_N\ (R\ x_N\ x_1 \ldots x_{N-1}))\ \textbf{in}\ (R\ x\ I \ldots I))$$

This series falls back to the level of I-chain$_N$.

Of course in the case of real programs, none of these tortuous examples can be encountered but, the first series may correspond to real parts of programs. For example, I-chain illustrates overlapped functions and Deep-let overlapped (recursively dependent) let instructions.

One can observe that the faster the algorithm is, the more easily the worst case complexity is reached.

5 Other analysis and future works

0CFA is a very popular but ambiguous analysis. The first definition of the 0CFA, found in Shiver's PLDI paper [18], corresponds to the LSR specification restricted to a language in CPS form. Due to the fact that all application nodes are in tail position, only two rules are needed. Whereas in his thesis [19], the specification is also called 0CFA but the analysis is based on an abstract semantics which needs to give an abstract environment to the approximation function. We believe that this specification is sound only in the case of CPS. Moreover, the soundness proof becomes harder[4]. Finally, direct implementation of the abstract semantics does not terminate and shortcuts are necessary to ensure termination.
Another analysis that could be performed on any λ-term, would consist in removing fix-point iterations but do not evaluate a body when no new value is added to formal parameter. In that case, we need to keep informations about free variables: a function body must be re-evaluated as soon as one of its free variables has changed. This analysis is more precise than LSR and seems to be a non-CPS version of 0CFA. Another perspective would consist in formally proving the soundness of this analysis and its complexity. Indeed, this seems to lead to more precise and less expensive analysis than most of the LSR analyses usually performed.
The differences between all these analyses, their exact complexity and a formal study of the relation between their results and LSR results is out of the scope of this paper. A promising perspective would consist in studying these differences, but this would require to add a CPS transformation to our study and is not directly related to this article.

6 Conclusion

In this article, we presented four different algorithms for calculating the Lowest Static Reduction. From simple (*blind*) implementation of the LSR rules, we specified an *evaluation-driven* algorithm taking nodes in a dedicated order similar to the evaluation one. This algorithm is control-flow oriented since it tries to follow paths used by the evaluator. Here, values are used as a whole by using set union and intersection.
Then, we optimized this algorithm by adding a virtual time allowing some shortcuts between fix-point iterations: the *time-sorted* algorithm avoids evaluating twice the same function on the same argument inside an iteration. This algorithm is also control-flow oriented in its outmost loops but each data is analyzed separately in its heart. The proof of the worst case complexity introduces the notion of *position*. This algorithm can be considered as a bridge between the *evaluation-driven* and the *position-driven* algorithms since the latter strongly depends on the position notion. This last algorithm is more data-flow oriented.
Providing fast operations on sets (constant time for checking the presence of one element in a set), the position-driven algorithm meets the cubic time complexity.

[4]Only 200 lines of Coq are needed to prove the soundness of the LSR, including the specification of the λ-calculus, the evaluator, the analysis and 45 lines of direct proofs.

As stated in section 2, LSR is intended to compute the *minimal* solution satisfying the rules given in 2.3. If we are not aware of minimality, a fast, linear and stupid algorithm will be: $\forall e \in \mathcal{E} : \ \mathcal{S}(e) \leftarrow \mathcal{L}$. One can easily check that the three rules given in 2.3 are satisfied. Between this useless algorithm and LSR, many algorithms can be studied. We hope that this work can be described in the same natural way as the position-driven algorithm.
The Coq proof of the soundness of LSR analysis, the Bigloo Scheme implementation of the four algorithms and the killer series can be found in:
`ftp://ftp-sop.inria.fr/oasis/Bernard.Serpette/jfla04.tar.gz`

6.1 Acknowledgements

The french version of this paper has been influenced by comments and remarks of anonymous referres. We would also like to thank Isabelle Attali and Andrew Ayers for their comments.

References

[1] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[2] A. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, Massachusetts Institute of Technology, September 1993.

[3] G. Barthe and B. Serpette. Static reduction analysis for imperative object oriented languages. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR'00*, volume 1955 of *Lecture Notes on Computer Science*, pages 344–361. Springer-Verlag, 2000.

[4] Dominique Boucher and Marc Feeley. Abstract compilation: a new implementation paradigm for static analysis. In *In T. Gyimothy, editor, Proceedings of CC'96, the 6th International Conference on Compiler Construction, number 1060 in Lecture Notes in Computer Science*, Linkoping, Sweden, April 1996. Springer-Verlag.

[5] N. Heintze. *Set based program analysis.* CMU-CS-92-201, Carnegie Mellon University, Pittsburgh, Pa, 1992.

[6] N. Heintze and D. McAllester. Control-flow analysis for ML in linear time. In *1997 ACM Conference on Programming Language Design and Implementation*, 1997.

[7] Nevin Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, pages 189–206, 1995.

[8] Nevin Heintze and David A. McAllester. Linear-time subtransitive control flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[9] Nevin Heintze and David A. McAllester. On the complexity of set-based analysis. In *International Conference on Functional Programming*, pages 150–163, 1997.

[10] Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science*, pages 342–351, 1997.

[11] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, 1995.

[12] Christian Mossin. Exact flow analysis. In *Fourth International Static Analysis Symposium (SAS)*, pages 250–264, Paris, France, 1997. Springer-Verlag.

[13] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. ESOP'01*, number 2028 in Lecture Notes in Computer Science, pages 252–268. Springer-Verlag, 2001.

[14] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

[15] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[16] B. Serpette. Approximations d'évaluateurs fonctionnels. In *Proceedings of WSA (Workshop on Static Analysis)*, pages 79–90. Bigre, 1992.

[17] M. Serrano and M. Feeley. Sorage Use Analysis and its Applications. In *1fst Int'l Conf. on Functional Programming*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[18] O. Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[19] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[20] O. Shivers. The Semantics of Scheme Control-Flow Analysis. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale University, New Haven, Conn., June 1991.

[21] Scott F. Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. *Lecture Notes in Computer Science*, 1782:382–??, 2000.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399